Programmieren von Klassen in Java

- Geschachtelte (innere) Klassen, Schnittstellen, Aufzählungen
- Mitglieds- oder Elementklassen
- Lokale Klassen

Innere Klassen

- Klassen, Aufzählungen, Schnittstelle können als Typen eingebettet werden (Innere Klassen)
- Motivation: Details zu verstecken, es gibt lokale Typdeklarationen, die keine Sichtbarkeit brauchen
- Klasse In, die in eine Klasse Out gesetzt wird:

```
class Out {
  class In {
  }
```

4 Typen von Inneren Klassen

- statische innere Klasse

```
class Out {
 static class In {}
```

- Mitgliedsklasse

```
class Out {
 class In { }
```

- lokale Klasse

```
class Out {
 Out() {
  class In { }
```

- anonyme innere Klasse

```
class Out {
 Out() {
  new Runnable() {
     public void run() { }
  };
```


Innere Klassen

- Gegenteil von geschachtelten Klassen heißt Top-Level-Klasse
- Die Laufzeitumgebung kennt nur Top-Level-Klassen
- Geschachtelte innere Klassen werden zu ganz »normalen« Klassendeklarationen

Statische innere Klassen und Schnittstellen

- *statische innere Klasse - nested top-level class*
- top-level: Klassen können das Gleiche können wie »normale« Klassen oder Schnittstellen (Unterpaket mit Namensraum)
- Zur Erzeugung von statischen inneren Klassen sind keine Objekte der äußeren Klasse nötig
- Die inneren (nicht statischen) Typen benötigen einen Verweis auf das äußere Objekt
- Oracle Spezifikation: die statischen inneren Klassen sind keine »echten« inneren Klassen

Statische innere Klassen und Schnittstellen

```
public class Lamp
{
  static String s = "Huhu";
  int i = 1;

  static class Bulb
  {
    void output()
    {
      System.out.println( s );
//    System.out.println( i );   // ☠ Compilerfehler: i is not static
    }
  }

  public static void main( String[] args )
  {
    Bulb bulb = new Lamp.Bulb();  // oder Lamp.Bulb bulb = ...
    bulb.output();
```

Statische innere Klassen und Schnittstellen

- *statische innere Klasse Bulb hat Zugriff auf alle anderen statischen Eigenschaften der äußeren Klasse Lamp, hier auf die Variable s.*
- *Zugriff auf Objektvariablen ist aus der statischen inneren Klasse nicht möglich (Klasse, im gleichen Paket)*
- *Der Zugriff von außen auf innere Klassen: ÄußereKlasse.InnereKlasse*
- *Innere Klasse muss anders heißen als die Äußere*

Statische innere Klassen und Schnittstellen

- Modifizierer *abstract, final* und Sichtbarkeitsmodifizierer
- Innere Klassen dürfen *public* oder *paketsichtbar*, alternativ aber auch *protected* oder *private* sein
- Eine *private statische innere Klasse* ist wie eine *private statische Variable*: sie kann nur von der umschließenden äußeren Klasse gesehen werden, aber nicht von anderen Top-Level-Klassen
- protected an statischen inneren Typen ermöglicht für den Compiler einen etwas effizienteren Bytecode

Innere Klassen

- Compiler generiert aus inneren Typen normale Klassendateien, mit synthetischen Methoden
- Der Compiler generiert neue Namen nach dem Muster: ÄußererTyp$InnererTyp, das heißt, ein Dollar-Zeichen trennt die Namen von äußerem und innerem Typ. Genauso heißt die entsprechende .class-Datei auf der Festplatte

Mitglieds- oder Elementklassen

- Mitgliedsklasse (engl. member class) ist vergleichbar mit einem Attribut

```
class House
{
  private String owner = "Ich";

  class Room
  {
    void ok()
    {
      System.out.println( owner );
    }
    // static void error() { }
```

Mitglieds- oder Elementklassen

- Klasse Room hat Zugriff auf alle Eigenschaften von House, auch auf die privaten
- Innere Mitgliedsklassen dürfen keine statischen Eigenschaften deklarieren. Der Versuch führt zu einem Compilerfehler:

```
The method error cannot be declared static;
```

```
static methods can only be declared in a static or top level type
```


Mitglieds- oder Elementklassen

- Um ein Exemplar von Room zu erzeugen, muss ein Exemplar der äußeren Klasse existieren
- Im Konstruktor oder in einer Objektmethode der äußeren Klassen kann mit dem new-Operator ein Exemplar der inneren Klasse erzeugt werden.
- Von außerhalb – oder von einem statischen Block der äußeren Klasse – muss bei Elementklassen sichergestellt sein, dass es ein Exemplar der äußeren Klasse gibt

Mitglieds- oder Elementklassen

- Java schreibt eine spezielle Form für die Erzeugung mit new vor, die folgendes allgemeine Format besitzt:

```
referenz.new InnereKlasse(...)
```

- referenz eine Referenz vom Typ der äußeren Klasse. Um in der statischen main()-Methode vom Haus ein Room-Objekt aufzubauen, schreiben wir:

```
House h = new House();
Room  r = h.new Room();
```

```
Room  r = new House().new Room();
```


Mitglieds- oder Elementklassen (this)

- Zugriff einer inneren Klasse In auf die this-Referenz der sie umgebenden Klasse Out: *Out.this*
- Überdecken Variablen der inneren Klasse die Variablen der äußeren Klasse, so wird Out.this.Eigenschaft geschrieben, um an die Eigenschaften der äußeren Klasse Out zu gelangen

Mitglieds- oder Elementklassen (this)

```
class FurnishedHouse
{
  String s = "House";

  class Room
  {
    String s = "Room";

    class Chair
    {
      String s = "Chair";

      void output()
      {
        System.out.println( s );
        System.out.println( this.s );
        System.out.println( Chair.this.s );
        System.out.println( Room.this.s );
        System.out.println( FurnishedHouse.this.s );
      }

      public static void main( String[] args )
      {
        new FurnishedHouse().new Room().new Chair().output();
      }
    }
  }
}
```

```
// Chair
// Chair
// Chair
// Room
// House
```


Mitglieds- oder Elementklassen (this)

- Objekte für die inneren Klassen Room und Chair lassen sich wie folgt erstellen

```
FurnishedHouse h            = new FurnishedHouse();
FurnishedHouse.Room r       = h.new Room();
FurnishedHouse.Room.Chair c = r.new Chair();
c.out();

// Exemplar von FurnishedHouse
// Exemplar von Room in h
// Exemplar von Chair in r
// Methode von Chair
```

Mitglieds- oder Elementklassen (this)

- FurnishedHouse.Room.Chair bedeutet nicht automatisch, dass FurnishedHouse ein Paket mit dem Unterpaket Room ist, in dem die Klasse Chair existiert.
- Die Doppelbelegung des Punkts verbessert die Lesbarkeit nicht, es droht Verwechslungsgefahr zwischen inneren Klassen und Paketen. Deshalb sollte die Namenskonvention beachtet werden: Klassennamen beginnen mit Großbuchstaben, Paketnamen mit Kleinbuchstaben

Mitglieds- oder Elementklassen (this)

- Für das Beispiel House und Room erzeugt der Compiler die Dateien House.class und House$Room.class.
- Damit innere Klasse an Attribute der äußeren gelangt, generiert der Compiler automatisch in jedem Exemplar der inneren Klasse eine Referenz auf das zugehörige Objekt der äußeren Klasse.
- Mit dem Referenz kann die innere Klasse auch auf nicht-statische Attribute der äußeren Klasse zugreifen.

Mitglieds- oder Elementklassen (this)

- Für die innere Klasse ergibt sich folgendes Bild in House$Room.class:

```
class HouseBorder$Room
{
  final House this$0;

  House$Room( House house )
  {
    this$0 = house;
  }
  // ...
```


Mitglieds- oder Elementklassen (this)

- Ist in einer Datei nur eine Klasse deklariert, kann diese nicht privat sein. Private innere Klassen sind aber legal.
- Statische Hauptklassen gibt es auch nicht, aber innere statische Klassen sind legitim.
- Die folgende Tabelle fasst die erlaubten Modifizier noch einmal kompakt zusammen:

Mitglieds- oder Elementklassen (this)

Modifizierer erlaubt auf	äußeren Klassen	inneren Klassen	äußeren Schnittstellen	inneren Schnittstellen
public	ja	ja	ja	ja
protected	nein	ja	nein	ja
private	nein	ja	nein	ja
static	nein	ja	nein	ja
final	ja	ja	nein	nein
abstract	ja	ja	ja	ja

Mitglieds- oder Elementklassen (this)

- Äußere Klassen können auf private Eigenschaften der inneren Klasse zugreifen

```
public class NotSoPrivate
{
  private static class Family { private String dad, mom; }

  public static void main( String[] args )
  {
    class Node { private Node next; }

    Node n = new Node();
    n.next = new Node();

    Family ullenboom = new Family();
    ullenboom.dad = "Heinz";
    ullenboom.mom = "Eva";
```

Mitglieds- oder Elementklassen (this)

- Innere Klasse kann auf alle Attribute der äußeren Klasse zugreifen. Eine innere Klasse wird als ganz normale Klasse übersetzt
- Eine innere Klasse kann auch auf private Eigenschaften zurückgreifen, eine Designentscheidung, die sehr umstritten ist und lange kontrovers diskutiert wurde. Doch wie?
- Der Trick ist, dass der Compiler eine synthetische statische Methode in der äußeren Klasse einführt:

Mitglieds- oder Elementklassen (this)

```
class House
{
  private String owner;

  static String access$0( House house )
  {
    return house.owner;
  }
```

Mitglieds- oder Elementklassen (this)

- Die statische Methode access$0() ist der Helfershelfer, der für ein gegebenes House das private Attribut nach außen gibt. Da die innere Klasse einen Verweis auf die äußere Klasse pflegt, gibt sie diesen beim gewünschten Zugriff mit, und die access$0()-Methode erledigt den Rest

- Für eine weitere private Variable int size, würde der Compiler ein int access$1(House) generieren.

Lokale Klassen

- Lokale Klassen sind innere Klassen,
- Sie werden direkt in Anweisungsblöcken von Methoden, Konstruktoren und Initialisierungsblöcken gesetzt
- Lokale Schnittstellen sind nicht möglich

Lokale Klassen

```
public class FunInside
{
  public static void main( String[] args )
  {
    int i = 2;
    final int j = 3;

    class In
    {
      In() {
          System.out.println( j );
//        System.out.println( i );    // Compiler error because i is not final
      }
    }
    new In();
  }
```


Lokale Klassen

- Deklaration der inneren Klasse In wie eine Anweisung.
- Sichtbarkeitsmodifizierer ist ungültig
- Keine Klassenmethoden und allgemeine statische Variablen (finale Konstanten ja).
- Kann auf Methoden der äußeren Klasse und auf finale lokale Variablen und Parameter zugreifen.
- Aus einer inneren statischen Methode, kann die lokale Klasse keine Objektmethoden der äußeren Klasse aufrufen

Anonyme innere Klassen

- Haben keinen Namen
- Erzeugen immer automatisch ein Objekt.
- Klassendeklaration und Objekterzeugung sind zu einem Sprachkonstrukt verbunden.
- Die allgemeine Notation ist folgende:

```
new KlasseOderSchnittstelle() { /* Eigenschaften der inneren Klasse */ }
```


Anonyme innere Klassen

- In den geschweifter Klammern werden:
  - Methoden und Attribute deklariert
  - Methoden überschrieben.
- new wird gefolgt vom Namen der Klasse oder Schnittstelle
- Keine *extends*- oder *implements*-Angaben und *eigene Konstruktoren* möglich.
- Objektmethoden und finale statische Variablen sind erlaubt

Anonyme innere Klassen

1. *new Klassenname(Optionale Argumente) { ... }*:
   - Anonyme Klasse ist eine Unterklasse von Klassenname
   - Argumente für den Konstruktor der Basisklasse notwendig, wenn z.B. die Oberklasse keinen Standardkonstruktor hat
2. *new Schnittstellenname() { ... }:*
   - anonyme Klasse erbt von Object
   - Ohne Implementierung der Schnittstellennoperationen (Fehler) läge eine abstrakte innere Klasse vor

Anonyme innere Klassen

Bp: Unterklasse von java.awt.Point, sie überschreibt die toString()-Methode

```
Point p = new Point( 10, 12 ) {
  @Override public String toString() {
    return "(" + x + "," + y + ")";
  }
};

System.out.println( p );    // (10,12)
```

Unterklasse von Point wird aufgebaut - Name der inneren Klasse fehlt.

Einziges Exemplar der anonymen Klasse lässt sich über die Variable p weiterverwenden.

Anonyme innere Klassen

- Der neue anonyme Typ hat eine Methode quote()
  - diese kann direkt aufgerufen werden
  - die quote()-Methode ist sonst unsichtbar, da Typ anonym
- Nur Methoden der Oberklasse (hier Object) beziehungsweise der Schnittstelle sind bekannt. (Eine Anwendung kann mit Reflection auf die Methoden zugreifen.)

Anonyme innere Klassen

- Für innere anonyme Klassen erzeugt der Compiler eine normale Klassendatei.
- Notation für Klassennamen: InnerToStringDate$1. Falls es mehr als eine innere Klasse gibt, folgen $2, $3 und so weiter (ÄußereKlasse$InnereKlasse geht wegen anonym nicht)

Anonyme innere Klassen

- Bp.: Für nebenläufige Programme, gibt es die Klasse Thread und die Schnittstelle Runnable:

  Schnittstelle Runnable hat Methode run(), diese wird in den parallel abzuarbeitenden Programmcode gesetzt. (mit einer inneren anonymen Klasse, die Runnable implementiert):

```
new Runnable() {     // Anonyme Klasse extends Object implements Runnable
  public void run() {
    ...
```


Anonyme innere Klassen

- Bp.: Exemplar kommt in den Konstruktor der Klasse Thread. Thread wird mit start() gestartet.

```
new Thread( new Runnable() {
  @Override public void run() {
    for ( int i = 0; i < 10; i++ )
      System.out.printf( "%d ", i );
  }
} ).start();

for ( int i = 0; i < 10; i++ )
  System.out.printf( "%d ", i );
```

- Ausgabe: `0 0 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9`

Anonyme innere Klassen

- Jede **anonyme Klasse** kann einen eigenen **Konstruktor** deklarieren (keine direkten) mit Hilfe von Exemplarinitialisierungsblöcke
- Anonyme Klassen können keinen direkten Konstruktor haben, Konstruktor gelangt über den Exemplarinitialisierer Programmcode in Bytecode-Datei.

Anonyme innere Klassen

Bp.: Die anonyme Klasse ist eine Unterklasse von Point und initialisiert im Konstruktor einen Punkt mit den Koordinaten –1, –1. Aus diesem speziellen Punkt-Objekt lesen wir dann die Koordinaten wieder aus:

```
java.awt.Point p = new java.awt.Point() { { x = -1; y = -1; } };

System.out.println( p.getLocation() );  // java.awt.Point[x=-1,y=-1]

System.out.println( new java.awt.Point( -1, 0 )
{
  {
    y = -1;
  }
}.getLocation() );                      // java.awt.Point[x=-1,y=-1]
```

Anonyme innere Klassen

- Im »anonymen Konstruktor« kein super()
- super() wird automatisch in den Initialisierungsblock eingesetzt
- Parameter für die gewünschte Variante des (überladenen) Oberklassen-Konstruktors werden am Anfang der Deklaration der anonymen Klasse angegeben. Zweites Beispiel:

```
System.out.println( new Point(-1, 0) { { y = -1; } }.getLocation() );
```


Anonyme innere Klassen

- Objekt BigDecimal wird initialisiert.
- Im Konstruktor der anonymen Unterklasse geben wir anschließend den Wert mit der geerbten toString()-Methode aus:

```
new java.math.BigDecimal( "12345678901234567890" ) {
  { System.out.println( toString() ); }
};
```

Anonyme innere Klassen

- Lokale und innere Klassen können auf lokalen Variablen (finale Parameter der umschließenden Methode lesend zugreifen)
- Veränderung mit Trick möglich. Zwei Lösungen:
  - Nutzung eines finalen Feldes der Länge 1, für das Ergebnis
  - Nutzung von AtomicXXX-Klassen aus dem java.util.concurrent.atomic-Paket, die ein primitives Element oder eine Referenz aufnehmen

Innere Klassen

- this in Unterklassen

```
public class Shoe
{
  void out()
  {
    System.out.println( "Ich bin der Schuh des Manitu." );
  }

  class LeatherBoot extends Shoe
  {
    void what()
    {
      Shoe.this.out();
    }
```

Innere Klassen

- this in Unterklassen

```
    @Override
    void out()
    {
      System.out.println( "Ich bin ein Shoe.LeatherBoot." );
    }
  }

  public static void main( String[] args )
  {
    new Shoe().new LeatherBoot().what();
  }
}
```